



# 创新创业实践课汇总报告

网络空间安全学院（研究院）

网安一班张雨萌

2023 年 08 月 02 日

## 目录

- (yes)\*Project1: implement the naïve birthday attack of reduced SM3
- (yes)\*Project2: implement the Rho method of reduced SM3
- (yes)\*Project3: implement length extension attack for SM3, SHA256, etc.
- (yes)\*Project4: do your best to optimize SM3 implementation (software)
- (yes)\*Project5: Impl Merkle Tree following RFC6962
- \*Project6: impl this protocol with actual network communication
- \*Project7: Try to Implement this scheme
- (yes)\*Project8: AES impl with ARM instruction
- (yes)\*Project9: AES / SM4 software implementation
- (yes)\*Project10: report on the application of this deduce technique in Ethereum with ECDSA
- (yes)\*Project11: impl sm2 with RFC6979
- (yes)\*Project12: verify the above pitfalls with proof-of-concept code
- (yes)\*Project13: Implement the above ECMH scheme
- (yes)\*Project14: Implement a PGP scheme with SM2
- (yes)\*Project15: implement sm2 2P sign with real network communication
- (yes)\*Project16: implement sm2 2P decrypt with real network communication
- (yes)\*Project17: 比较 Firefox 和谷歌的记住密码插件的实现区别
- \*Project18: send a tx on Bitcoin testnet, and parse the tx data down to every bit, better write script yourself
- (yes)\*Project19: forge a signature to pretend that you are Satoshi
- (yes)\*Project20: ECMH PoC
- (yes)\*Project21: Schnorr Bacth
- (yes)\*Project22: research report on MPT

**\*Project1: implement the naïve birthday attack of reduced SM3**

## # 一、代码说明

生日攻击的目的是寻求一个基于 sm3 哈希值的弱碰撞，原理是一定长度和 hash 值结果  $2^{32}$  长度，在  $2^{16}$  密文空间中可以以 50% 以上的概率找到一个 hash 碰撞。这里使用了类似查表攻击似的数据结构，一边存表一边查表（可以使用多线程进一步优化脚本性能），以便可以在较短时间内找到一个前 16bit 的 hash 弱碰撞。如果寻找更长 bit 的碰撞，寻找时间也会相应变长。

## # 二、运行指导

将源码 clone 到本地运行 main 函数即可运行生日攻击脚本。

## #三、软件环境：Visual Studio 2022

硬件环境：处理器 11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz 2.61 GHz（过后不再赘述）

## #四、部分代码：

//查表寻找弱碰撞

```
for (int j = 0; j < i; j++) {
    if (outlist[j].substr(0, 4) == result.substr(0, 4)) {
        cout << endl;
        cout << "collision string input 1 :" + str << endl << endl;
        cout << "collision hash value 1:" << endl;
        cout << result.substr(0, 8) << " ";
        cout << result.substr(8, 8) << " ";
        cout << result.substr(16, 8) << " ";
        cout << result.substr(24, 8) << " ";
        cout << result.substr(32, 8) << " ";
        cout << result.substr(40, 8) << " ";
        cout << result.substr(48, 8) << " ";
        cout << result.substr(56, 8) << " ";
        cout << endl;
        cout << "collision string input 2 :" + inlist[j] << endl << endl;
        cout << "collision hash value 2:" << endl;
        cout << outlist[j].substr(0, 8) << " ";
        cout << outlist[j].substr(8, 8) << " ";
        cout << outlist[j].substr(16, 8) << " ";
        cout << outlist[j].substr(24, 8) << " ";
        cout << outlist[j].substr(32, 8) << " ";
        cout << outlist[j].substr(40, 8) << " ";
        cout << outlist[j].substr(48, 8) << " ";
        cout << outlist[j].substr(56, 8) << " ";
        cout << endl << "finding num in all: " << i;
        return;
    }
}
```

## \*Project2: implement the Rho method of reduced SM3

### #一、步骤

1. 随机生成 1 个消息
2. 由前面消息的 hash 作为新的消息，并求 hash 值
3. 判断是否构成环

### #二、攻击原理

Rho Method 是一种随机化算法，每一个数都由前一个数决定，可以生成的数是有限的，所以会进入循环。因为 Rho Method 是一种随机化算法，所以每次寻找碰撞的时间也不确定。

### #三、代码摘要

```

project2.py - D:\py呀呀呀呀\project2.py (3.8.5)
File Edit Format Run Options Window Help

def rho_method():
    # 生成随机数
    x = str(random.randint(0, pow(2, bin_length)))
    # x_a = x_1
    x_a = sm3.sm3_hash(func.bytes_to_list(bytes(x, encoding='utf-8')))
    # x_b = x_2
    x_b = sm3.sm3_hash(func.bytes_to_list(bytes(x_a, encoding='utf-8')))
    # 记录原像
    msg1 = x
    msg2 = x_a
    while x_a[:hex_length] != x_b[:hex_length]:
        # x_a = x_i
        msg1 = x_a
        x_a = sm3.sm3_hash(func.bytes_to_list(bytes(x_a, encoding='utf-8')))
        # x_b = x_{i+1}
        temp = sm3.sm3_hash(func.bytes_to_list(bytes(x_b, encoding='utf-8')))
        x_b = sm3.sm3_hash(func.bytes_to_list(bytes(temp, encoding='utf-8')))
        msg2 = temp
    print("找到碰撞,hash值为-->{}".format(x_a[:hex_length]))

    print("原像为:")
    print("消息1:", msg1)
    print("消息1hash值", sm3.sm3_hash(func.bytes_to_list(bytes(msg1, encoding='u
    print()
    print("消息2:", msg2)
    print("消息2hash值", sm3.sm3_hash(func.bytes_to_list(bytes(msg2, encoding='u

if __name__ == '__main__':
    print("{}bits hash值碰撞!".format(bin_length))

    start = time.time()
    rho_method()
    end = time.time()

    print("为了找到碰撞所花费的时间:{}".format(end - start) + "s")

```

### #四、运行结果

```

===== RESTART: D:\py呀呀呀呀\project2.py =====
=====
16bits hash值碰撞!
找到碰撞,hash值为-->b320
原像为:
消息1: f8bdf451a18a6bf3a135bd92299572a92285061e66c5d41a0f4b1a67d6047015
消息1hash值 b320f3be8589b37b2f7bb98c6c44a8378184792f443829958acd84aeab9e343a
消息2: d32e28fd663aa17a310987245135bb4ac1ffc28b9c7735b38f8b4abca6acb54f
消息2hash值 b32093a96f2c3fbd44ad39abd76cfb70a0e45df9303429f0159cc10b5a9c4560
为了找到碰撞所花费的时间:175.74261450767517s
>>> |

```

**\*Project3: implement length extension attack for SM3, SHA256, etc.**

## #一、步骤:

1. 首先计算原消息(secret)的 hash 值
2. 在 secret+padding 之后附加一段消息,用原消息的 hash 值作为 IV 计算附加消息之后的 hash 值,得到消息扩展后的 hash1
3. 用 sm3 加密伪造后的整体消息,得到 hash2
4. 验证 hash1 与 hash2 是否相等

## #二、攻击原理

SM3 的消息长度是 64 字节或者它的倍数,如果消息的长度不足则需要 padding。在 padding 时,首先填充一个 1,随后填充 0,直到消息长度为 56(或者再加整数倍的 64)字节,最后 8 字节用来填充消息的长度。在 SM3 函数计算时,首先对消息进行分组,每组 64 字节,每一次加密一组,并更新 8 个初始向量(初始值已经确定),下一次用新向量去加密下一组,以此类推。我们可以利用这一特性去实现攻击。当我们得到第一次加密后的向量值时,再人为构造一组消息用于下一次加密,就可以在不知道 secret 的情况下得到合法的 hash 值。

## #三、代码摘要

```

project3.py - D:\py呀呀呀呀\project3.py (3.8.5)
File Edit Format Run Options Window Help
##### SM3的实现 #####

# ff函数
def ff(x, y, z, j):
    # x,y,z分别为3个32位向量(int), j为轮数(0<=j<64)
    if 0 <= j < 16:
        ret = x ^ y ^ z
    elif 16 <= j < 64:
        ret = (x & y) | (x & z) | (y & z)
    return ret

# gg函数
def gg(x, y, z, j):
    # x,y,z分别为3个32位向量(int), j为轮数(0<=j<64)
    if 0 <= j < 16:
        ret = x ^ y ^ z
    elif 16 <= j < 64:
        ret = (x & y) | ((~x) & z)
    return ret

def shift_left(x, n):
    # x是32位的向量
    strx = bin(x)[2:].zfill(32)
    add = '0' * n
    strret = strx[n:] + add
    ret = int(strret, 2)
    return ret

# p1置换(消息扩展)
def p1(x):
    ret = x ^ (shift_left(x, 15)) ^ (shift_left(x, 23))
    return ret

# p0置换(压缩函数)
def p0(x):
    ret = x ^ (shift_left(x, 9)) ^ (shift_left(x, 17))
    return ret

```



```

# 压缩函数cf
def sm3_cf(vi, bi):
    # 消息扩展
    w = []
    # w列表都是十进制整数
    # 生成w0-w15
    for i in range(16):
        wei = 0x1000000 # 权重 初始为2^24
        rel = 0
        for j in range(i * 4, (i + 1) * 4): # 将4字节数整合成一个字
            rel = rel + bi[j] * wei # bi[j]数据类型是二进制对应的整数, 比如0b101--5
            wei = int(wei / 0x100) # 权重每次减少2^8
        w.append(rel)

    # 生成w16-w67
    for k in range(16, 68):
        tmp = p1(w[k - 16] ^ w[k - 9] ^ shift_left(w[k - 3], 15)) ^ shift_left(w[k - 13], 7)
        w.append(tmp)
    w1 = []
    # 生成w' 0-w'64
    for k in range(0, 64):
        tmp = w[k] ^ w[k + 4]
        w1.append(tmp)
    A, B, C, D, E, F, G, H = vi
    # 64轮加密
    for j in range(0, 64):
        #
        ssl = shift_left((shift_left(A, 12)) ^ E ^ (shift_left(T[j], j % 32)), 7)
        #
        ss2 = ssl ^ shift_left(A, 12)
        tt1 = ff(A, B, C, j) ^ D ^ ss2 ^ w1[j]
        tt2 = gg(E, F, G, j) ^ H ^ ssl ^ w[j]
        D = C
        C = shift_left(B, 9)
        B = A
        A = tt1
        H = G
        G = shift_left(F, 19)
        F = E
        E = p0(tt2)
    v_i = [A, B, C, D, E, F, G, H]
    return [v_i[i] ^ vi[i] for i in range(0, 8)]

```

#### #四、运行结果



## Python 3.8.5 Shell

—      □      ×

File Edit Shell Debug Options Window Help

```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
```

>>>

```
===== RESTART: D:\py呀呀呀呀\project3.py =====
```

原消息: Hello, world!

原消息16进制: 48656c6c6f2c776f726c6421

原消息hash值: 8b405c03b70dd559effd42d701a20400275246ace5f5273a970dee4db1730e4e

附加消息: Welcome to cst!

附加消息的16进制: 57656c636f6d6520746f2063737421

构造的消息的hash值: 50061de46d2940e5eddecad7f33a8a00799214af57b890b9b715ee4dee6b0e4e

扩展后新消息的16进制：48656c6c6f2c776f722d6364218000

[illegible]

扩展后新消息的hash值: 50061de46d2940e5eddecad7f33a8a00799214af57b890b9b715ee4dee6b0e4e

攻击成功!

>>>

**\*Project4: do your best to optimize SM3 implementation (software)**

## #一、实验思路

## 1. 消息处理优化

(1) 初始化时, 消息扩展只计算  $W_0-W_3$  四个 32-bit 字

(2) 在优化后的轮函数中首先计算  $W[i+4]$ , 然后再计算  $W'[i]=W[i]^W[i+4]$

经过这样的调整, 去掉了字  $W'_0, \dots, W'_{63}$ , 减少了字  $W_0, \dots, W_{67}$  和  $W'_0, \dots, W'_{63}$  的加载和存储次数, 提高了消息扩展的速度

## 2. 每一轮压缩函数优化:

(1) 为了减少循环移位导致的不必要的赋值运算, 可以将字的循环右移变更每轮输入字顺序的变动, 且这个顺序变动会在 4 轮后还原

(2) 优化压缩函数的中间变量的生成流程, 去除不必要的赋值, 减少中间变量个数

(3) 利用上述调整以及消息扩展部分的调整可以将原来计算 TT1、TT2、D 和 H 的过程进行进一步简化

## #二、部分代码:

*/\*sm3 主函数: 包括消息填充、扩展、压缩步骤, 返回最后的 hash 值\*/*

```
std::vector<uint32_t> SM3::sm3_hash_optimize()
{
    caculT(); //预计算压缩函数中要用到的  $T[i] \ll i$  的值
    std::vector<uint32_t> hash_result(32, 0); // (0, 0, 0, ..., 0) 32 个 0, 用来存放 hash 结果
    unsigned char buffer[4] = { 0x01, 0x03, 0x4a, 0x95 }; //要处理的明文
    unsigned char hash_output[32]; //存放结果的中间变量
    SM3::SM3_optimize(buffer, 4, hash_output); //sm3 的主函数, 包括消息填充、扩展、压缩函数
    hash_result.assign(&hash_output[0], &hash_output[32]); //要返回一个 vector 变量所以把值赋到
    hash_result
    return hash_result;
}
```

## \*Project5: Impl Merkle Tree following RFC6962

### #一、代码说明

以 sm3 的散列模块为基础，用 C++ 代码实现了默克尔树，可以手动输入数据，默克尔树叶节点会生成相应的哈希值，非叶节点会生成相应的联结哈希值。

可以运行 test.cpp 对该模块的构建默克尔树功能进行简单测试。

### #二、运行指导

将源码 clone 到本地运行 main 函数即可运行。

### #三、软件环境：Visual Studio 20

### #五、部分代码

//默克尔树类

```
template<class T>
class MerkleTree {
private:
    MerkleTreeNode<T>* root;
public:
    //构造函数
    MerkleTree() {
        root = NULL;
        cout << "Now start to construct the MerkleTree" << endl;
        CreateNode(root);
    }
    //析构函数
    ~MerkleTree() {
        root = NULL;
    }
    //前序构建默克尔树
    void CreateNode(MerkleTreeNode<T>*& treeNode) {
        cout << "Please enter value or '#' to stop:";
        T value;
        cin >> value;
        treeNode = new MerkleTreeNode<T>(value);
        if (value == "#") {
            delete treeNode;
            treeNode = NULL;
        }
        else {
```



```

        CreateNode(treeNode->leftchild);
        CreateNode(treeNode->rightchild);
        //如果不是叶子结点，则对左右两个字结点做联合哈希
        if (treeNode->leftchild != NULL && treeNode->rightchild != NULL) {
            treeNode->data = iteration(padding(treeNode->leftchild->data +
treeNode->rightchild->data));
        }
    }
}

//取根节点
MerkleTreeNode<T>* getRoot() {
    return root;
}

//递归前序遍历
void PreOrder(MerkleTreeNode<T>* treeNode) {
    if (treeNode != NULL) {
        treeNode->VisitNode();
        PreOrder(treeNode->getLeft());
        PreOrder(treeNode->getRight());
    }
}

//递归中序遍历
void InOrder(MerkleTreeNode<T>* treeNode) {
    if (treeNode != NULL) {
        InOrder(treeNode->getLeft());
        treeNode->VisitNode();
        InOrder(treeNode->getRight());
    }
}

//递归后序遍历
void PostOrder(MerkleTreeNode<T>* treeNode) {
    if (treeNode != NULL) {
        PostOrder(treeNode->getLeft());
        PostOrder(treeNode->getRight());
        treeNode->VisitNode();
    }
}
};

```

**\*Project6: impl this protocol with actual network communication**

用实际的网络通信实现该协议

**\*Project7: Try to Implement this scheme**

试图实施这个计划

**\*Project8: AES impl with ARM instruction**

```
.section .data
.align 4
key:
    .word 0x2B7E1516, 0x28AED2A6, 0xABF71588, 0x09CF4F3C
text:
    .word 0x3243F6A8, 0x885A308D, 0x313198A2, 0xE0370734
.section .text
.global _start
_start:
    @ Load key into registers
    LDR r0, =key
    LDR r1, [r0]          @ r1 = key[0]
    LDR r2, [r0, #4]      @ r2 = key[1]
    LDR r3, [r0, #8]      @ r3 = key[2]
    LDR r4, [r0, #12]     @ r4 = key[3]
    @ Load text into registers
    LDR r0, =text
    LDR r5, [r0]          @ r5 = text[0]
    LDR r6, [r0, #4]      @ r6 = text[1]
    LDR r7, [r0, #8]      @ r7 = text[2]
    LDR r8, [r0, #12]     @ r8 = text[3]
    @ Round 0 -- Add Round Key
    ADD r5, r5, r1        @ r5 ^= r1
    ADD r6, r6, r2        @ r6 ^= r2
    ADD r7, r7, r3        @ r7 ^= r3
    ADD r8, r8, r4        @ r8 ^= r4
    mov r9, #10           @ r9 = 10 (number of rounds)
    mov r10, #1           @ round counter
round_loop:
    @ SubBytes
    bl sub_bytes
    @ ShiftRows
    bl shift_rows
    @ MixColumns
```

```

    bl  mix_columns
    @ AddRoundKey
    bl  add_round_key
    @ Check if this is the last round
    cmp r10, r9
    beq end_encryption
    @ Update round counter
    add r10, r10, #1
    b   round_loop
end_encryption:
    @ Store the result back
    LDR r0, =text
    STR r5, [r0]
    STR r6, [r0, #4]
    STR r7, [r0, #8]
    STR r8, [r0, #12]
    @ Terminate the program
    MOV r7, #1
    SWI 0x11
sub_bytes:
    @ Implement SubBytes logic here
    @ (omitted for brevity)
    bx lr
shift_rows:
    @ Implement ShiftRows logic here
    @ (omitted for brevity)
    bx lr
mix_columns:
    @ Implement MixColumns logic here
    @ (omitted for brevity)
    bx lr
add_round_key:
    @ Implement AddRoundKey logic here
    @ (omitted for brevity)
    bx lr
.end

```

## \*Project9: AES / SM4 software implementation

Feistel 网络和 SP 网络是分组密码设计中的两种不同的整体结构类型，也是使用最多的结构类型。AES 使用的是 SP 网络。

Feistel 网络（又称 Feistel 结构）把某一函数（通常称为 F 函数，又称轮函数）转化为一个置换。由于 Horst Feistel 在设计分组密码时发明的，所以用了这位发明者的名字，并且因为 DES 的使用而流行。当前世界上著名密码中还有 Twofish、MARS、FEAL、GOST、LOKI、E2、Blowfish、Camellia 和 RC6 等是采用 Feistel 结构的。（Feistel 结构还分平衡 Feistel 和非平衡 Feistel，这里只讨论平衡 Feistel。） 对于一个分组长度为  $2n$  bits 的  $r$  轮 Feistel 型分组密码，其加密过程如下：

Step1: 给定明文  $X$ ，记  $X = L_0 R_0$ ，其中  $L_0$  是  $X$  的左边  $n$  bits， $R_0$  是  $X$  的右边  $n$  bits。

Step2: 进行  $r$  轮循环运算。根据下列规则计算  $L_i R_i$ ， $1 \leq i \leq r$ ：

$L_i = R_{i-1}$   $R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$ ，这里， $F: F_{2n} \times F_{2N} \rightarrow F_{2n}$  是轮函数， $K_1, \dots, K_r$  是由种子密钥  $K$  生成的子密钥， $N$  为子密钥的长度。

Step3: 输出密文  $C = R_r L_r$

在加密的最后一轮，为了使算法同时用于加密盒解密，略去“左右交换”。

SP 网络（又称 SP 结构，Substitution & Permutation Net, SPN）是 Feistel 网络的一种推广，Rijndael、SAFER 和 SHARK 等著名密码算法都采用此结构。在这种密码结构中，每轮的输入首先经过一个由子密钥控制的可逆函数  $S$ ，然后再被作用于一个置换（或一个可逆的线性变换） $P$ 。 $S$  被称为混淆层，主要起混淆的作用； $P$  被称为扩散层，主要起扩散的作用。

## \*Project10:report on the application of this deduce technique in Ethereum with ECDSA

### #一、实验内容

主要完成了以下 4 个 task

1. Leaking  $k$  leads to leaking of  $d$  ( $k$  的泄露会导致泄露  $d$ )
2. Reusing  $k$  leads to leaking of  $d$  (对不同的消息使用相同的  $k$  进行签名会泄露  $d$ )
3. Two users, using  $k$  leads to leaking of  $d$ , that is they can deduce each other's  $d$  (两个不同的 user 使用相同的  $k$ , 可以相互推测对方的私钥  $d$ )
4. Malleability, e.g.  $(r, s)$  and  $(r, -s)$  are both valid signatures (验证  $(r, s)$  and  $(r, -s)$  均为合法签名)

其中 1 和 2 还分别根据推测得到的  $d$  值进行了消息的伪造并成功通过验签

### #二、文件内容

1. ECDSA.py: 包含了 ECDSA 密钥生成、签名、验签算法
2. pitfalls.py: 完成 4 个 task

### #三、攻击原理

1. Leaking  $k$  leads to leaking of  $d$  ( $k$  的泄露会导致泄露  $d$ )

由签名算法中  $s = k^{-1} * (e + dr) \bmod n$  推得  $d = (s * k - e) / r$

2. Reusing  $k$  leads to leaking of  $d$  (对不同的消息使用相同的  $k$  进行签名会泄露  $d$ )

$$s_1 = k^{-1} * (e_1 + dr) \bmod n$$

$$s2 = k^{-1} * (e2 + dr) \bmod n$$

推得  $s1/s2 = (e1 + dr) / (e2 + dr)$  即  $d = ((e1 - e2) * s2 / (s1 - s2) - e2) * r^{-1} \bmod n$

3. Two users, using  $k$  leads to leaking of  $d$ , that is they can deduce each other's  $d$  (两个不同的 user 使用相同的  $k$ , 可以相互推测对方的私钥  $d$ )

两个不同的 user 使用相同的  $k$ : 此类情况下相当于 leaking  $k$ , 所以可以相互推测对方的私钥  $d$

4. Malleability, e.g.  $(r, s)$  and  $(r, -s)$  are both valid signatures (验证  $(r, s)$  and  $(r, -s)$  均为合法签名)

首先生成签名  $(r, s)$ , 然后使用验签算法验证  $(r, -s)$  是否是合法签名

#### #四、代码摘要

# ECDSA 签名, 并返回签名以及  $k$

```
def ECDSA_sign_and_return_k(m, sk):
    """ECDSA signature algorithm
    :param m: message
    :param sk: private key
    :return signature: (r, s), k
    """
    while 1:
        k = secrets.randbelow(N) # N is prime, then k <- Zn*
        R = EC_multi(k, G)
        r = R[0] % N # Rx mod n
        if r != 0: break
    e = sm3.sm3_hash(func.bytes_to_list(bytes(m, encoding='utf-8')))) # e = hash(m)
    e = int(e, 16)
    s = (inv(k, N) * (e + sk * r) % N) % N
    return (r, s), k
```

# 使用给定的  $k$  进行签名

```
def ECDSA_sign_and_assign_k(m, k, sk):
    """ECDSA signature algorithm
    :param m: message
    :param sk: private key
    :return signature: (r, s), k
    """
    R = EC_multi(k, G)
    r = R[0] % N # Rx mod n
    e = sm3.sm3_hash(func.bytes_to_list(bytes(m, encoding='utf-8')))) # e = hash(m)
    e = int(e, 16)
    tmp1 = inv(k, N)
    tmp2 = (e + sk * r) % N
    s = tmp1 * tmp2 % N
    return (r, s), k
```

```

# 椭圆曲线加法
def EC_add(p, q):
    # 0 means inf
    if p == 0 and q == 0:
        return 0 # 0 + 0 = 0
    elif p == 0:
        return q # 0 + q = q
    elif q == 0:
        return p # p + 0 = p
    else:
        if p[0] == q[0]:
            if (p[1] + q[1]) % P == 0:
                return 0 # mutually inverse
            elif p[1] == q[1]:
                return EC_double(p)
        elif p[0] > q[0]: # swap if px > qx
            tmp = p
            p = q
            q = tmp
        r = []
        slope = (q[1] - p[1]) * inv(q[0] - p[0], P) % P # 斜率
        r.append((slope ** 2 - p[0] - q[0]) % P)
        r.append((slope * (p[0] - r[0]) - p[1]) % P)
        return (r[0], r[1])

def EC_inv(p):
    """椭圆曲线逆元"""
    r = [p[0]]
    r.append(P - p[1])
    return r

# 椭圆曲线减法: p - q
def EC_sub(p, q):
    q_inv = EC_inv(q)
    return EC_add(p, q_inv)

```

#五、运行结果？

File Edit Shell Debug Options Window Help

Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>>

===== RESTART: D:\py呀呀呀呀\project10 ECDSA.py =====

>>>

===== RESTART: D:\py呀呀呀呀\project10 pitfalls.py =====

>>>

Traceback (most recent call last):

File "D:\py呀呀呀呀\project10 pitfalls.py", line 10, in <module>

from ECDSA import \*

ModuleNotFoundError: No module named 'ECDSA'

>>> |



## \*Project11: impl sm2 with RFC6979

### #一、实验内容

本 code 实现 sm2 的加解密算法以及签名验签算法

### #二、实现原理

按照 ppt 所给出的 SM2 Parameters、SM2 Signature Algorithm、SM2 Verify Algorithm、SM2 Encryption、SM2 Decryption 进行实现

### #三、代码摘要

```
# SM2加密
def SM2_enc(M, pk):
    """SM2 encryption algorithm
    :param M: message (str)
    :param pk: public key
    :return: ciphertext, C1:dot C2:hex_str C3hex_str

    if pk == 0:
        return 'error:public key = 0!'
    while 1:
        k = secrets.randbelow(N)
        C1 = EC_multi(k, G) # C1 = kG = (x1, y1)
        dot = EC_multi(k, pk) # kpk = (x2, y2)
        klen = get_bit_num(M)
        x2 = hex(dot[0])[2:]
        y2 = hex(dot[1])[2:]
        t = KDF(x2 + y2, klen)
        # t = 0 is invalid
        if (t != '0' * klen):
            break
    C2 = enc_XOR(M, t)
    temp = bytes((x2 + M + y2), encoding='utf-8')
    C3 = sm3.sm3_hash(func.bytes_to_list(temp))
    return (C1, C2, C3)

# SM2解密
def SM2_dec(C, sk):
    """SM2 decryption algorithm
    :param C: (C1, C2, C3)
    :param sk: private key
    :return: plaintext

    C1, C2, C3 = C
    # 验证C1是否满足曲线方程
    x, y = C1
    left = pow(y, 2) % P
    right = (pow(x, 3, P) + A * x + B) % P
    if (left != right):
        return "Error:C1 get error!"
    # 计算椭圆曲线点S
    S = h * C1
    if S == 0:
        return 'Error:S=0 get error!'
    # 计算[ds]C1 = (x2,y2)
    dot = EC_multi(sk, C1)
    klen = len(C2) * 4
    x2 = hex(dot[0])[2:]
    y2 = hex(dot[1])[2:]
    t = KDF(x2 + y2, klen)
    if t == '0' * klen:
        return "Error:t = 0!"
    # 计算M' = C2 xor t
    M = dec_XOR(C2, t)
    temp = bytes((x2 + M + y2), encoding='utf-8')
    u = sm3.sm3_hash(func.bytes_to_list(temp))
    if u != C3:
        return "Error: u != C3!"
    return M
```

```

# SM2签名
def sm2_sign(sk, msg, ZA):
    """SM2 signature algorithm"""
    gangM = ZA + msg
    gangM_b = bytes(gangM, encoding='utf-8')
    e = sm3.sm3_hash(func.bytes_to_list(gangM_b))
    e = int(e, 16) # str -> int
    while 1:
        k = secrets.randbelow(N) # generate random number k
        a_dot = EC_multi(k, G) # (x1, y1) = kG
        r = (e + a_dot[0]) % N # r = (e + x1) % n
        s = 0
        if r != 0 and r + k != N:
            s = (inv(1 + sk, N) * (k - r * sk)) % N
        if s != 0: return (r, s)

# SM2验签
def sm2_verify(pk, ID, msg, signature):
    """SM2 verify algorithm
    :param pk: public key
    :param ID: ID
    :param msg: message
    :param signature: (r, s)
    :return: true/false
    """
    r = signature[0] # r'
    s = signature[1] # s'
    ZA = precompute(ID, A, B, G_X, G_Y, pk[0], pk[1])
    gangM = str(ZA) + msg
    gangM_b = bytes(gangM, encoding='utf-8')
    e = sm3.sm3_hash(func.bytes_to_list(gangM_b)) # e'
    e = int(e, 16) # str -> int
    t = (r + s) % N

    dot1 = EC_multi(s, G)
    dot2 = EC_multi(t, pk)
    dot = EC_add(dot1, dot2) # (x2, y2) = s'G + t'pk

    R = (e + dot[0]) % N # R = (e' + x2) % N
    return R == r

```

#### #四、运行结果

```

>>>
===== RESTART: D:\py呀呀呀呀\project11.py =====
验证加解密
-----
plaintext为: sdu_cst_project.
-----
\ciphertext为:
C1: (24675288848171207331599881205984527181732120889048128511046812291616830276813, 6822095997693671
152991262845832223708928518523961708359228425316722053107482)
C2 53df14bbb103f28b26426f8dc1785eb7
C3: 599c79e67cc55d11abb631f1645db57337423013d121c75abc38ace3d2e2398c
-----
plaintext from ciphertext为: sdu_cst_project.
-----
是否成功解密: True
-----
验证签名
-----
signature为:
(10840175751879966296412620533895972958773235059171841086499756468708810239710, 60040981685227786003
586889330693614524715067960655371682730786205927775216454)
-----
signature是否合法: True
\
\
\

```

**\*Project12: verify the above pitfalls with proof-of-concept code**

## #一、poc 脚本的编写步骤

1. 首先新建一个.py 文件，文件名应当符合命名格式

2. 编写 poc 实现类 DemoPOC，继承自 POCBase 类

```
1 from pocsuit3.api import Output, POCBase, register_poc, requests, logger
2 from pocsuit3.api import Output, get_listener_ip, get_listener_port
3 from pocsuit3.api import REVERSE_PAYLOAD # REVERSE_PAYLOAD 反弹shell模块;
4 from pocsuit3.lib.utils import random_str
5
6
7 class DemoPOC(POCBase):
```

3. 填写 poc 信息字段，需要认真填写所有字段的基本信息，规范信息以便于查找

```
1 class DemoPOC(POCBase): # 实现类DemoPoc,继承自POCBase
2     vulID = '1.1' # ssvid ID ,如果是提交漏洞的同时提交POC, 则写成
3     version = '1' # 默认为1
4     author = ['1'] # POC作者的名字
5     vulDate = '2021-2-2' # 漏洞公开的时间, 不明确时可以写当天
6     createDate = '2020/10/10' # 编写POC日期
7     updateDate = '1.1' # poc更新日期, 默认和编写时间一样
8     references = ['vulhub'] # 漏洞地址来源, 0day不用写
9     name = 'flask-poc' # poc名称
10    appPowerLink = 'flask' # 漏洞厂商的主页地址
11    appName = 'flask' # 漏洞应用名称
12    appVersion = 'flask' # 漏洞影响版本
13    vulType = VUL_TYPE.CODE_EXECUTION # 漏洞类型
14    # vulType定义了poc的种类, VUL_TYPE属于enums.py种的一个类; CODE_EXECUTION = 'Code Execution' #代码执行
15    desc = ''' #此处是对poc的描述, 展开数组时key带入SQL语句形成SQL注入, 可以添加管理员, 伪装成信息泄露
16        flask
17    ...
18    samples = ['96.234.71.117:80'] # 利用poc举例的一个地址, 漏洞的简要描述
19    install_requires = [] # 测试样例, 使用poc测试成功的网站,
```

4. 编写验证模式，在\_verify 方法中写入 POC 雁阵脚本

```
1 def _verify(self):
2     output = Output(self) # 验证代码
3     if result: # result表示返回结果
4         output.success(result)
5     else:
6         output.fail('target is not vulnerable')
7     return output
```

5. 编写攻击模式。用\_attack() 函数中，写入 EXP 脚本，在攻击模式下可以对目标进行 getshell，查询管理员账户密码等操作，定义它的方法与检测模式类似：

```
1 def _attack(self):
2     output = Output(self)
3     result = {}
4     #攻击代码
```

**\*Project13: Implement the above ECMH scheme**

#一、原始思路:将 hash 值作为椭圆曲线上的 x 点,根据椭圆曲线方程  $y^2 = x^3 + Ax + B$  来计算 y 值,但是会存在无解的情况,所以舍弃该做法

#二、本 code 将消息 hash 到椭圆曲线上的方法

- (1) 首先计算消息的 sm3 hash 值
- (2) 将 sm3 hash 值作为 k 值
- (3)  $(x, y) = k * G$

#三、本 code 完成的测试

- (1)  $\text{hash}(\{a, b\}) == \text{hash}(\{b, a\})$
- (2)  $\text{hash}(a) + \text{hash}(b) == \text{hash}(\{a, b\})$
- (3)  $\text{hash}(\{a, b, c\}) - \text{hash}(c) == \text{hash}(\{a, b\})$
- (4)

#四、代码摘要

```
# 将消息hash到椭圆曲线上
def hash_to_dot(msg):
    # 将sm3的hash值作为x点
    x = sm3.sm3_hash(func.bytes_to_list(bytes(msg, encoding='utf-8'))))
    x = int(x, 16) % N
    # 求y
    # hash为椭圆曲线上的点
    hash_value = EC_multi(x, G)
    return hash_value

# Elliptic curve MultiSet Hash ----- Combine/add/remove elements
# combine操作:生成集合的hash值
def combine(msg_set):
    hash_set = EC_add(0, 0)
    for i in range(len(msg_set)):
        hash_set = EC_add(hash_set, hash_to_dot(msg_set[i]))
    return hash_set

# add操作:hash({a, b}) + hash(c) --- 集合的hash值 + 单个消息的hash值
def add(hash_set, msg):
    hash_value = hash_to_dot(msg)
    hash_set = EC_add(hash_set, hash_value)
    return hash_set

# remove操作:hash({a, b, c}) - hash(c) --- 集合的hash值 - 单个消息的hash值
def remove(hash_set, msg):
    hash_value = hash_to_dot(msg)
    hash_set = EC_sub(hash_set, hash_value)
    return hash_set
```

#五、运行结果



Python 3.8.5 Shell

— □ ×

File Edit Shell Debug Options Window Help

Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

&gt;&gt;&gt;

===== RESTART: D:\py呀呀呀呀\project13.py =====

计算a, b, c的hash值

hash(a):

(9442326346504103285762355240969705231896309482403450070685949813255577511062, 96390389297051152515459084300943067875390047810489553542863220555642396594702)

hash(b):

(49519791566470678336790150671875406611520633228298079447694021028002621921066, 105115900778494145928703966810283677507601945384176789254615254579861315166880)

hash(c):

(113559246783006880066880716993033308665177196049642916611783036245079186046759, 65023036978586174845790607607060905626110848082664460080019787734781227953111)

测试hash({a, b}) == hash({b, a})

hash({a, b}):

(107846338683800133077939168378697419452237791276878572724718471398597475115511, 76361589148707743227854522443639865431327211335455867500311042164702967004152)

hash({b, a}):

(107846338683800133077939168378697419452237791276878572724718471398597475115511, 76361589148707743227854522443639865431327211335455867500311042164702967004152)

hash({a, b}) == hash({b, a})?: True

测试add操作: hash(a) + hash(b) == hash({a, b})

hash(a) + hash(b):

(107846338683800133077939168378697419452237791276878572724718471398597475115511, 76361589148707743227854522443639865431327211335455867500311042164702967004152)

hash({a, b}):

(107846338683800133077939168378697419452237791276878572724718471398597475115511, 76361589148707743227854522443639865431327211335455867500311042164702967004152)

hash(a) + hash(b) == hash({a, b})? True

测试sub操作: hash({a, b, c}) - hash(c) == hash({a, b})

hash({a, b, c}):

(10131672057849182152062493509270559823293931543347558562093300656645222200561, 90284060903266374613924122080083579232478962929716109352689243787145972948550)

hash({a, b, c}) - hash(c):

(107846338683800133077939168378697419452237791276878572724718471398597475115511, 76361589148707743227854522443639865431327211335455867500311042164702967004152)

hash({a, b}):

(107846338683800133077939168378697419452237791276878572724718471398597475115511, 76361589148707743227854522443639865431327211335455867500311042164702967004152)

hash({a, b, c}) - hash(c) == hash({a, b})? True

&gt;&gt;&gt;

## \*Project14: Implement a PGP scheme with SM2

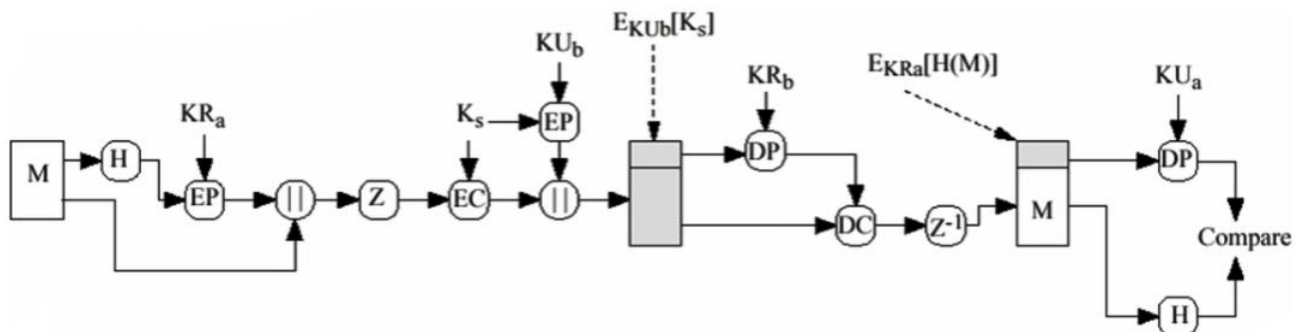
### #一、PGP 概念

PGP (Pretty Good Privacy) 加密，由一系列散列、数据压缩、对称密钥加密，以及公钥加密的算法组合而成，每个步骤支持几种算法。PGP 支持消息认证和完整性检测：

- ①完整性检测被用来检查消息在传输过程中是否变更过（即验证消息完整性）；
- ②消息认证则是被用来决定消息是否确由某特定的人或实体发出（即数字签名验证）。

在 PGP 中，这些特性默认是和消息加密同时开启的，而且同样可以被应用到明文的验证。发送者只需使用 PGP 为消息创建一个数字签名，即以数据或信息创建一个散列，然后使用发送者的私钥利用散列生成数字签名。

### #二、PGP 加密流程



### #三、举例说明

假设在一个数据 M 传输的过程中，发送者为 A、接受者为 B，公私钥对 (pk, rk)，会话密钥 s k 散列函数（哈希函数）用 H 代替，公钥密码算法以 R S A 为例，对称加密算法以 A E S 为例：

1、数据经过散列处理 H 形成摘要：

$H(M)$

2、用公钥密码算法 R S A RSARSA 对摘要数字签名：

$sig_{RSA-rk_A}(H(M))$

3、链接数据与数字签名得到证书，（并进行压缩，压缩可省略）：

$sig_{RSA-rk_A}(H(M)) || M$

4、对称加密算法 A E S AESAES 加密前一步得到的证书：

$AES_{sk}(sig_{RSA-rk_A}(H(M)) || M)$

5、公钥密码算法 R S A RSARSA 构造数字信封，封装密钥：

$RSA_{pk_B}(sk) || RSA_{pk_B}(AES_{sk}(sig_{RSA-rk_A}(H(M)) || M))$

6、链接，完成 PGP 加密：

$RSA_{pk_B}(sk) || AES_{sk}(sig_{RSA-rk_A}(H(M)) || M)$

除了上述利用 H、R S A、A E S H、RSA、AES、RSA、AES 的 PGP 加密原理应用，还可以用 S M 2、S M 3、S M 4 SM2、SM3、SM4SM2、SM3、SM4 构建一个同样具有机密性和认证服务的文件传输方案：

$SM2_{pk_B}(sk) || SM4_{sk}(sig_{SM2-rk_A}(SM3(M))) || M$



## #四、代码摘要

```

# SM2加密
def SM2_enc(M, pk):
    """SM2 encryption algorithm
    :param M: message (str)
    :param pk: public key
    :return: ciphertext, C1:dot C2:hex_str C3hex_str
    """
    if pk == 0:
        return 'error:public key = 0!'
    while 1:
        k = secrets.randbelow(N)
        C1 = EC_multi(k, G) # C1 = kG = (x1, y1)
        dot = EC_multi(k, pk) # kpk = (x2, y2)
        klen = get_bit_num(M)
        x2 = hex(dot[0])[2:]
        y2 = hex(dot[1])[2:]
        t = KDF(x2 + y2, klen)
        # t = 0 is invalid
        if (t != '0' * klen):
            break
    C2 = enc_XOR(M, t)
    temp = bytes((x2 + M + y2), encoding='utf-8')
    C3 = sm3.sm3_hash(func.bytes_to_list(temp))
    return (C1, C2, C3)

# SM2解密
def SM2_dec(C, sk):
    """SM2 decryption algorithm
    :param C: (C1, C2, C3)
    :param sk: private key
    :return: plaintext
    """
    C1, C2, C3 = C
    # 验证C1是否满足曲线方程
    x, y = C1
    left = pow(y, 2) % P
    right = (pow(x, 3, P) + A * x + B) % P
    if (left != right):
        return "Error:C1 get error!"
    # 计算椭圆曲线点S
    S = h * C1
    if S == 0:
        return 'Error:S=0 get error!'
    # 计算[ds]C1 = (x2,y2)
    dot = EC_multi(sk, C1)
    klen = len(C2) * 4
    x2 = hex(dot[0])[2:]
    y2 = hex(dot[1])[2:]
    t = KDF(x2 + y2, klen)
    if t == '0' * klen:
        return "Error:t = 0!"
    # 计算M' = C2 xor t
    M = dec_XOR(C2, t)
    temp = bytes((x2 + M + y2), encoding='utf-8')
    u = sm3.sm3_hash(func.bytes_to_list(temp))
    if u != C3:
        return "Error: u != C3!"
    return M

```

**\*Project15: implement sm2 2P sign with real network communication**

## #一、文件内容

1. SM2.py:包含 SM2 加解密算法以及签名验签算法
2. server.py:server 端
3. client.py:client 端

#运行指导:在 pycharm 中先运行 server.py, 然后运行 client.py

## #二、实验内容

implement sm2 2P sign with real network communication

公钥: $P = [(d1d2)^{-1} - 1] * G$

私钥: $d = (d1d2)^{-1} - 1$

对于消息, Server 与 Client 需要协作才能进行签名, 因为二者均不能独自计算得到私钥 image

## (一)Server 端:

1. 接收 Client 端  $P1 = d1^{-1} * G$
2. 生成私钥  $d2$ , 计算公钥  $P = [(d1d2)^{-1} - 1] * G$  并公开公钥  $P$
3. 与 client 进行交互共同签名:接收 client 发送的  $Q1, e$  并返回  $r, s2, s3$  给 client

## (二)Client 端:

1. 生成私钥  $d1$  以及  $P1 = d1^{-1} * G$ , 将  $P1$  发送给 server
2. 与 server 进行交互共同签名:
  - (1) 规定两方 ID 以及消息  $msg$
  - (2) 计算  $Q1, e$  并发送给 server
  - (3) 接收  $r, s2, s3$  并计算最终签名  $(r, s)$

## #三、代码摘要

 project15 Server端.py - D:\py呀呀呀\project15 Server端.py (3.8.5) — □ ×

File Edit Format Run Options Window Help

## # 实现与client的交互

```
def interact():
    server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server.bind(('', 8090))
    while 1:
        # 接收P1 = d1^-1 * G以及client IP
        P1, addr = receive_P1(server)
        # 生成私钥d以及公钥P = [(d1d2)^-1 - 1] * G
        d2, P = d_and_P(P1)
        # 公开公钥P, 将公钥发给client
        print("公钥为:")
        print(P)
        print("-----")
        data = str(P[0]) + ',' + str(P[1])
        server.sendto(data.encode(), addr)
        # 接收Q1, e
        Q1, e, addr = receive_Q1_e(server)
        # 计算r, s2, s3
        r, s2, s3 = comp_r_s2_s3(d2, Q1, e)
        # 发送r, s2, s3给client
        data = str(r) + ',' + str(s2) + ',' + str(s3)
        server.sendto(data.encode(), addr)
    s.close()
```

project15 Client端.py - D:\py呀呀呀呀\project15 Client端.py (3.8.5) — □

File Edit Format Run Options Window Help

```
# 实现与server的交互
def interact():
    client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # 生成私钥与P1 = d^-1 * G
    d1, P1 = d_and_P()
    data = str(P1[0]) + ',' + str(P1[1])
    # 将P1 = d^-1 * G发送给server
    client.sendto(data.encode(), ("127.0.0.1", 8090))
    # 接收公钥P
    P = receive_P(client)
    # 发送Q1, e
    k1 = send_Q1_e(client)
    # 接收r, s2, s3
    r, s2, s3 = receive_r_s2_s3(client)
    # 生成最终签名
    signature = create_sign(d1, k1, r, s2, s3)
    r, s = signature
    print("签名为:")
    print("r:", r)
    print("s:", s)
    client.close()
```

# 四、运行结果

Python 3.8.5 Shell — □

File Edit Shell Debug Options Window Help

```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit
D64] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:\py呀呀呀呀\project15 Client端.py =====
>>>
Traceback (most recent call last):
  File "D:\py呀呀呀呀\project15 Client端.py", line 14, in <module>
    from SM2 import *
ModuleNotFoundError: No module named 'SM2'
>>>
```

**\*Project16: implement sm2 2P decrypt with real network communication**

## #一、文件内容

1. SM2.py:包含 SM2 加解密算法以及签名验签算法
2. server.py:server 端
3. client.py:client 端
4. 运行指导:在 pycharm 中先运行 server.py, 然后运行 client.py

## #二、实验内容

implement sm2 2P decrypt with real network communication

公钥: $P = [(d1d2)^{-1} - 1] * G$

私钥: $d = (d1d2)^{-1} - 1$

对于密文, Server 与 Client 需要协作才能进行解密, 因为二者均不能独自计算得到私钥 image

## #三、交互流程

## (一)Server 端:

1. 接收 Client 端  $P1 = d1^{-1} * G$
2. 生成私钥  $d2$  并计算公钥  $P = [(d1d2)^{-1} - 1] * G$
3. 这里 Server 端自己生成密文并将密文返回给 client
4. 与 client 进行交互共同解密:接收 client 发送的  $T1$  并返回  $T2$  给 client

## (二)Client 端:

1. 生成私钥  $d1$  以及  $P1 = d1^{-1} * G$
2. 接收密文
3. 与 server 进行交互共同解密:
  - (1) 检查  $C1 \neq 0$  并且计算  $T1 = d1^{-1} * C1$
  - (2) 接收  $T2$  并恢复明文

## #四、代码摘要

## # 实现与client的交互

```
def interact():
    server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server.bind('', 13800)
    while 1:
        # 接收P1 = d1^-1 * G以及client IP
        P1, addr = receive_P1(server)
        # 生成私钥d以及公钥P = [(d1d2)^-1 - 1] * G
        d2, P = d_and_P(P1)
        # 对消息进行加密,生成密文
        msg = "Zhang_SDU_201900180019"
        print("Client需要恢复的明文消息为:", msg)
        ciphertext = SM2_enc(msg, P)
        # 将密文发送给client
        data = str(ciphertext)
        server.sendto(data.encode(), addr)
        # 接收T1, 计算T2并发送给client
        recv_T1_and_comp_T2(d2, server)
        print("-----")
    server.close()
```



```

# 实现与server的交互
def interact():
    client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # 生成私钥与P1 = d^-1 * G
    dl, P1 = d_and_P()
    data = str(P1[0]) + ',' + str(P1[1])
    # 将P1 = d^-1 * G发送给server
    client.sendto(data.encode(), ("127.0.0.1", 13800))
    # 接收密文
    data, addr = client.recvfrom(4096)
    data = eval(data)
    C1, C2, C3 = data
    print("接收的密文为:")
    print("C1:", C1)
    print("C2:", C2)
    print("C3:", C3)
    print("-----")
    # 检查C1 != 0并且计算T1 = d1^-1 * C1
    T1 = check_and_comp(dl, C1)
    # 将T1发送给server
    data = str(T1[0]) + ',' + str(T1[1])
    client.sendto(data.encode(), addr)
    # 接收T2
    data, addr = client.recvfrom(1024)
    data = data.decode()
    index1 = data.index(',')
    T2 = (int(data[:index1]), int(data[index1 + 1:]))
    # 由密文恢复明文消息
    plaintext = recover_msg(T2, C1, C2, C3)
    print("恢复的明文为:", plaintext)
    print("-----")
    client.close()

```

## # 五、运行结果

```

C:\Windows\system32\cmd.exe
File "C:\Users\DELL\AppData\Local\Programs\Python\Python38\lib\site-packages\pip\_internal\network\download.py", line
147, in _call
    for chunk in chunks:
File "C:\Users\DELL\AppData\Local\Programs\Python\Python38\lib\site-packages\pip\_internal\cli\progressBars.py", line
53, in _rich_progress_bar
    for chunk in iterable:
File "C:\Users\DELL\AppData\Local\Programs\Python\Python38\lib\site-packages\pip\_internal\network\utils.py", line 63,
in response_chunks
    for chunk in response.raw.stream(
File "C:\Users\DELL\AppData\Local\Programs\Python\Python38\lib\site-packages\pip\_vendor\urllib3\response.py", line 62
2, in stream
    data = self.read(amt=amt, decode_content=decode_content)
File "C:\Users\DELL\AppData\Local\Programs\Python\Python38\lib\site-packages\pip\_vendor\urllib3\response.py", line 58
7, in read
    raise IncompleteRead(self._fp_bytes_read, self.length_remaining)
File "C:\Users\DELL\AppData\Local\Programs\Python\Python38\lib\contextlib.py", line 131, in __exit__
    self.gen.throw(type, value, traceback)
File "C:\Users\DELL\AppData\Local\Programs\Python\Python38\lib\site-packages\pip\_vendor\urllib3\response.py", line 44
3, in _error_catcher
    raise ReadTimeoutError(self._pool, None, "Read timed out.")
pip._vendor.urllib3.exceptions.ReadTimeoutError: HTTPSConnectionPool(host='files.pythonhosted.org', port=443): Read time
d out.
C:\Users\DELL>

```

## \*Project17: 比较 Firefox 和谷歌的记住密码插件的实现区别

### #一、Firefox 记住密码插件

浏览器一般都具有自动记录密码功能,但是这往往也会给我们带来一些的开发上的困扰,比如火狐浏览器,首先抛出问题:

前端加密一般是:对密码框内的密码进行加密,然后又把加密后的值塞回密码框中,并通过 form 表单提交到后端进行解密,验证登录。

如果是火狐浏览器,它会监控 post 动作,询问是否记录账号密码,以便下次登录自动填充,但这时火狐浏览器保存的却是加密后的密码,导致下次打开登录页面自动填充了加密后的密码,再次点击登录密码会二次加密,从而登录失败!

经过验证,Chrome 会很机智的记住加密前的密码,没有这个问题,怎么解决火狐出现的这种情况,首先先看看火狐的自动记住密码和自动填充的过程:

1. 火狐浏览器自动记住密码是在监控到 post 登录请求后,从上往下,找到最后一个 type= 'password' 的 input 框,然后询问你是否记住该密码;

2. 火狐浏览器自动填充是从上而下,寻找第一个为 type= 'password' 的 input 框,并把值填充进去,不管这个 input 是否是隐藏 (display:none) 的。

结合以上分析过程,我们就能解决自动填充的问题:

1. 把原来的密码框 A 设置成隐藏域类型,即 type= 'hidden', 隐藏起来,前端看不见,其余 id, name 属性不变;

2. 在原来的密码框 A 下面增加一个密码框 B: `<input type='password' id='passwordTemp' >`, PS: 此处不要设置 name 属性, form 表单就不会提交;

3. 在前端登录按钮的 click() 方法中执行如下逻辑:

(1) . 获取 B 的值 value, js 加密后得到 encryptedValue, 并把 encryptedValue 赋值给 A, 然后 submit (此时, 表单把 A 的值提交到后端);

(2) . 执行 submit 后, 将 encryptedValue 再赋值给 B, 这样前端密码框就会有密码变长的加密效果, 而且加密后的密码是在 post 过后赋值的, 火狐浏览器记住的是 post 时 B 的未加密密码;

这样, 我们就能保证,

1. post 提交的密码是加密的, 不会有被抓包解析出来的危险;

2. 火狐也会自动记录正确未加密的密码;

3. 密码自动填充, 也会准确找到显示 B 密码框, 不会填充到 A 中, 导致每次还要手动选择密码;

### #二、firefox 记住密码插件部分源码

```
<?xml version="1.0" encoding="UTF-8"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:em="http://www.mozilla.org/2004/em-rdf#">
  <Description about="urn:mozilla:install-manifest">
    <em:id>firesheep@codebutler.com</em:id>
    <em:type>2</em:type>
    <em:name>Firesheep</em:name>
    <em:version>0.1</em:version>
    <em:creator>Eric Butler</em:creator>
    <em:contributor></em:contributor>
    <em:aboutURL>chrome://firesheep/content/about.xul</em:aboutURL>
    <em:optionsURL>chrome://firesheep/content/preferences/prefsWindow.xul</em:optionsURL>
```



```

<em:homepageURL>http://codebutler.github.com/firesheep</em:homepageURL>
<em:updateURL>http://codebutler.github.com/firesheep/update.rdf</em:updateURL>

<em:updateKey>MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCx0jVpi2A7WnvIDJlqYF1Sk+9Ny6ccmRSFShlcpmjQfEGYDJFN
LnXERSHhNGIVYArBhHkunWwYzJ4Es4kUZ6BGE0IbEIPhJPgD18DCdgRwFL+ts9sn+2DWxd0KrU62P9phCNcsvr1L+G+0/zNsoBYDei
MS00rGN5HcftwdReHaMQIDAQAB</em:updateKey>
  <em:unpack>true</em:unpack>
  <em:targetPlatform>Darwin_x86-gcc3</em:targetPlatform>
  <em:targetPlatform>WINNT_x86-msvc</em:targetPlatform>
  <em:targetApplication>
    <Description>
      <em:id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</em:id> <!-- Firefox -->
      <em:minVersion>3.6.10</em:minVersion>
      <em:maxVersion>3.6.*</em:maxVersion>
    </Description>
  </em:targetApplication>
</Description>
</RDF>

```

### #三、谷歌记住密码插件

ShowPassword 是一款能够显示出当前密码框中的密码的谷歌浏览器插件，在用户输入密码的时候，如果忘记了当前已经输入的密码，或者不确定输入的密码是否正确的前提下，就可以使用 ShowPassword 插件来临时查看一下当前的密码。

还有一种情况是，用户如果使用的谷歌浏览器的记住密码功能，在网站上输入密码的地方，谷歌浏览器会自动帮助用户填上帐号和密码，但是久而久之，用户就会因为长时间没有操作过，而忘记当前网站中的密码，这时候，如果使用 ShowPassword 插件，就可以临时查看一下当前密码框中的密码，来给用户一个提醒的作用。

ShowPassword 的使用方法：

1. 在谷歌浏览器中安装 ShowPassword 插件，并在 Chrome 的扩展器中启动显示密码的功能，ShowPassword 插件的下载地址可以在本文的下方找到，离线 ShowPassword 插件的安装方法可参考：怎么在谷歌浏览器中安装.crx 扩展名的离线 Chrome 插件？
2. 安装完成以后，用户就可以使用 Chrome 打开需要输入帐号和密码的网站，在密码框中输入一定的字符以后，把鼠标放在密码框 500 毫秒以后，该密码框就会把密码以明文的方式显示出来

**\*Project18: send a tx on Bitcoin testnet, and parse the tx data down to every bit, better write script yourself**

**\*Project19: forge a signature to pretend that you are Satoshi**

## #一、实验内容

本实验实现了当 ECDSA 验签算法在不验证  $m$  的情况下, 能够通过公钥来伪造合法签名

## #二、攻击原理

1. 在  $\mathbb{F}_n^*$  上随机选择  $u, v$ , 计算  $R = (x, y) = uG + vP$  ( $P$  为公钥)
2.  $r = x \bmod n$
3.  $e = ruv^{-1} \bmod n$
4.  $s = rv^{-1} \bmod n$
5.  $\text{signature} = (r, s)$

## #三、攻击前提

验签算法: 不验证消息  $m$

$$e * s^{-1} * G + r * s^{-1} * P = kG = R$$

前提数据: Satoshi 的公钥, 这里我们通过随机生成来代替 Satoshi 的真实公钥

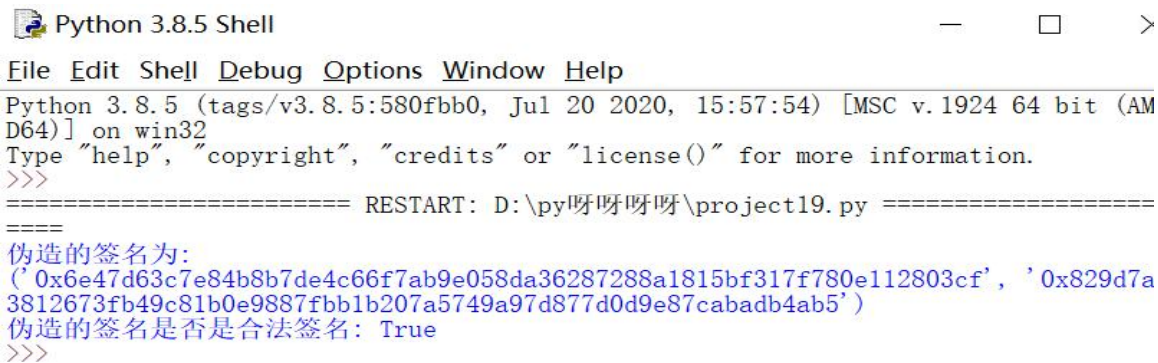
## 四、代码摘要

```
# ECDSA签名
def ECDSA_sign(m, sk):
    """ECDSA signature algorithm
    :param m: message
    :param sk: private key
    :return signature: (r, s)
    """
    while 1:
        k = secrets.randbelow(N) # N is prime, then k <- Zn*
        R = EC_multi(k, G)
        r = R[0] % N # Rx mod n
        if r != 0:
            break
    e = sm3.sm3_hash(func.bytes_to_list(bytes(m, encoding='utf-8')))) # e = hash(m)
    e = int(e, 16)
    s = inv(k, N) * (e + sk * r) % N
    return (r, s)

# ECDSA验签:不使用message
def ECDSA_verify_m_not_check(signature, e, pk):
    r, s = signature
    x = EC_multi(inv(s, N), EC_add(EC_multi(e, G), EC_multi(r, pk)))
    return x[0] % N == r

# 伪造中本聪的签名
def Pretend_Satoshi(pk):
    u = random.randrange(1, N - 1)
    v = random.randrange(1, N - 1)
    R = EC_add(EC_multi(u, G), EC_multi(v, pk))
    r = R[0] % N
    e = (r * u * inv(v, N)) % N
    s = (r * inv(v, N)) % N
    signature_forge = (r, s)
    print("伪造的签名为:")
    print((hex(r), hex(s)))
    return ECDSA_verify_m_not_check(signature_forge, e, pk)
```

## #五、运行结果



```
Python 3.8.5 Shell
File Edit Shell Debug Options Window Help
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:\py呀呀呀呀\project19.py =====
=====
伪造的签名为:
('0x6e47d63c7e84b8b7de4c66f7ab9e058da36287288a1815bf317f780e112803cf', '0x829d7a3812673fb49c81b0e9887fbb1b207a5749a97d877d0d9e87cabadb4ab5')
伪造的签名是否是合法签名: True
>>>
```

## \*Project20: ECMH PoC

## #一、概念

PoC 即 Proof of Capacity，也就是容量证明。PoC 也被称作空间证明，是 PoW 共识机制的升级版。主要就是通过数据预生成后检索的算法，这使得整个系统比 PoW 更加快速，PoC 仅需在数分钟内生成新区块，而 PoW 需要 10 分钟。相比 PoW 是通过计算来获取区块的打包权，PoC 则是将“计算”变成了“寻找”。

整个记账权的争取过程如下：

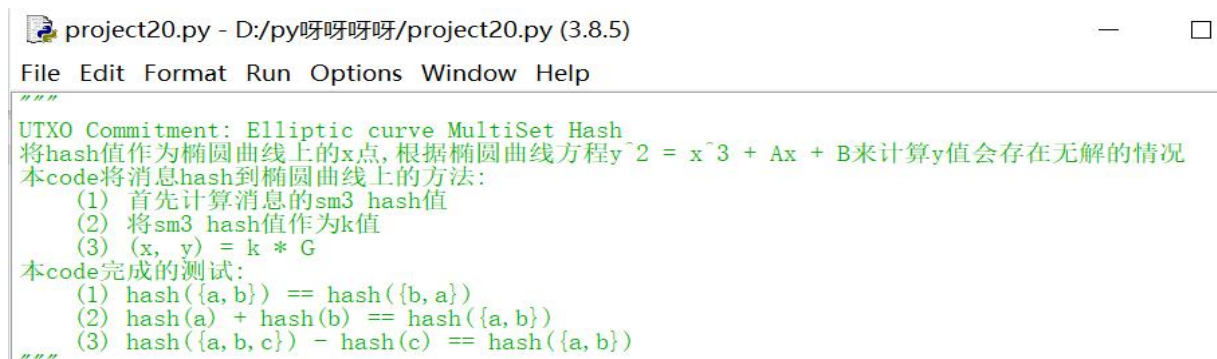
在挖矿还没开始前，网络就会把破解谜题的计算方法（solutions）储存在硬盘空间里。这些计算方法有的比较快，有的比较慢，如果你的硬盘里恰好有一个计算方法，是目前最近产生的这个区块里的谜题所对应的“最快解”，那么你就赢得了这个区块的记账权——挖矿的奖励就是你的了。如果你在硬盘空间里拥有越多的计算方法（也被称为 plots），你用最快的速度去破解当前这个区块的谜题，实现的概率也就越大。



PoC 共识机制下的挖矿被称为硬盘挖矿，从最早的 Burst 到大火的 IPFS，再到 7 月的 BHD，还有现在的 BSN，每一个 PoC 共识下的项目都会被市场追捧和追随。BSN 被誉为 PoC 算法中的新比特。

PoC 共识机制虽然不具备完全取代 PoW 挖矿的可能性，但是，它的确是代表了矿业新的探索方向。

## #二、一些实践



```
project20.py - D:\py呀呀呀呀\project20.py (3.8.5)
File Edit Format Run Options Window Help
"""
UTXO Commitment: Elliptic curve MultiSet Hash
将hash值作为椭圆曲线上的x点,根据椭圆曲线方程y^2 = x^3 + Ax + B来计算y值会存在无解的情况
本code将消息hash到椭圆曲线上的方法:
(1) 首先计算消息的sm3 hash值
(2) 将sm3 hash值作为k值
(3) (x, y) = k * G
本code完成的测试:
(1) hash({a, b}) == hash({b, a})
(2) hash(a) + hash(b) == hash({a, b})
(3) hash({a, b, c}) - hash(c) == hash({a, b})
"""
```



```

Python 3.8.5 Shell
File Edit Shell Debug Options Window Help
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:/py呀呀呀呀/project20.py =====
计算a, b, c的hash值
hash(a):
(9442326346504103285762355240969705231896309482403450070685949813255577511062, 963903
89297051152515459084300943067875390047810489553542863220555642396594702)
hash(b):
(49519791566470678336790150671875406611520633228298079447694021028002621921066, 10511
5900778494145928703966810283677507601945384176789254615254579861315166880)
hash(c):
(113559246783006880066880716993033308665177196049642916611783036245079186046759, 6502
3036978586174845790607607060905626110848082664460080019787734781227953111)
测试hash({a, b}) == hash({b, a})
hash({a, b}):
(107846338683800133077939168378697419452237791276878572724718471398597475115511, 7636
1589148707743227854522443639865431327211335455867500311042164702967004152)
hash({b, a}):
(107846338683800133077939168378697419452237791276878572724718471398597475115511, 7636
1589148707743227854522443639865431327211335455867500311042164702967004152)
hash({a, b}) == hash({b, a})?: True
测试add操作: hash(a) + hash(b) == hash({a, b})
hash(a) + hash(b):
(107846338683800133077939168378697419452237791276878572724718471398597475115511, 7636
1589148707743227854522443639865431327211335455867500311042164702967004152)
hash({a, b}):
(107846338683800133077939168378697419452237791276878572724718471398597475115511, 7636
1589148707743227854522443639865431327211335455867500311042164702967004152)
hash(a) + hash(b) == hash({a, b})? True
测试sub操作: hash({a, b, c}) - hash(c) == hash({a, b})
hash({a, b, c}):
(10131672057849182152062493509270559823293931543347558562093300656645222200561, 90284
060903266374613924122080083579232478962929716109352689243787145972948550)
hash({a, b, c}) - hash(c):
(107846338683800133077939168378697419452237791276878572724718471398597475115511, 7636
1589148707743227854522443639865431327211335455867500311042164702967004152)
hash({a, b}):
(107846338683800133077939168378697419452237791276878572724718471398597475115511, 7636
1589148707743227854522443639865431327211335455867500311042164702967004152)
hash({a, b, c}) - hash(c) == hash({a, b})? True
>>>

```

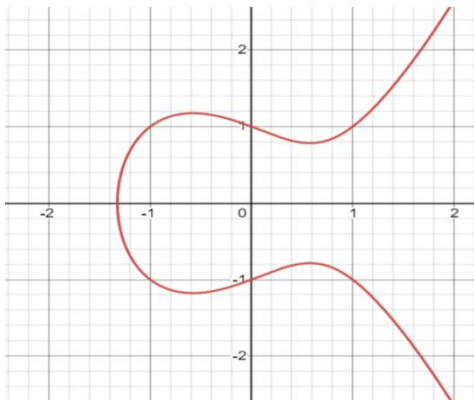
## \*Project21: Schnorr Bacth

### 一、基本知识

Schnorr 签名算法最初是由德国密码学家 Claus Schnorr 于 2008 年提出的，在密码学中，它是一种数字签名方案，以其简单高效著称，其安全性基于某些离散对数问题的难处理性。

密码学中，常用以下形式的椭圆曲线： $E: y^2 = x^3 + ax + b \pmod{p}$  同时要求  $4a^3 + 27b^2 \neq 0$ 。其中  $p$  为一个大素数， $a$ 、 $b$ 、 $x$  和  $y$  均在有限域  $GF(p)$  中，即从  $\{0, 1, \dots, p-1\}$  中取值。该曲线常用  $E_{\{p\}}(a, b)$  表示。若该曲线上只有有限个离散点，设为  $N$  个，则椭圆曲线的阶为  $N$ 。 $N$  越大，椭圆曲线安全性越高。椭圆曲线的阶可通过 Schoof 算法计算求得。

椭圆曲线  $E: y^2 = x^3 - x + 1$  图形如下：



### 加法规则

椭圆曲线  $E_{\{p\}}(a, b)$  在如下定义的法法规则构成 Abel 群（交换群）。

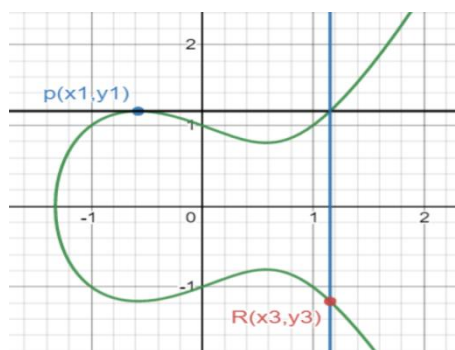
$$O + O = O$$

$\forall P = (x, y) \in E_{\{p\}}(a, b)$ , 有  $P + O = O + P = P$

$\forall P = (x, y) \in E_{\{p\}}(a, b)$ , 有  $P + (-P) = O$ ,  $P$  的逆为  $-P = (x, -y)$

$\forall P = (x_1, y_1), Q = (x_2, y_2) \in E_{\{p\}}(a, b)$ , 则  $P + Q = R = (x_3, y_3) \in E_{\{p\}}(a, b)$

其中  $x_3 = \lambda^2 - x_1 - x_2, y_3 = \lambda(x_1 - x_3) - y_1$



### 乘法规则

$\forall k \in \mathbb{Z}, \forall P \in E_p(a, b)$ , 有  $kP = P + \dots + P$  ( $k$  个  $P$  相加);

$\forall s, t \in \mathbb{Z}, \forall P \in E_p(a, b)$ , 有  $(s+t)P = sP + tP, s(tP) = (st)P$

## 二、算法细节

**2.1 公私钥产生算法 (KeyGen) :**

- 选择一条椭圆曲线  $E_p(a, b)$  和基点  $G$ ;
- 选择私钥  $d_A$  ( $d_A < n$ ,  $n$  为该  $G$  的阶) , 利用基点  $G$  计算公钥  $Q_A = d_A \cdot G$ ;

**2.2 签名生成算法 (Sign)**

- 选择一个随机整数  $k$  ( $k < n$ );
- 计算点  $R = k \cdot G = (x_1, y_1)$ ;
- 计算  $\sigma = k + \text{hash}(m || R) \cdot d_A \pmod n$ ;
- 得到签名  $s = (R, \sigma)$ ;

**2.3 签名验证算法 (Verify) :**

- 验证等式:  $\sigma \cdot G \equiv \text{hash}(m || R) \cdot Q_A + R$ ;
- 如果等式成立输出1, 否则输出0。

Schnorr 签名除了上面一种形式外, 还有另外一种形式:

**2.4 签名生成算法 (Sign)**

- 选择一个随机整数  $k$  ( $k < n$ );
- 计算点  $R = k \cdot G = (x_1, y_1)$ ;
- 计算  $\alpha = \text{hash}(m || R)$
- 计算  $\sigma = k + \text{hash}(m || R) \cdot d_A \pmod n$ ;
- 得到签名  $s = (\alpha, \sigma)$ ;

**2.5 签名验证算法 (Verify) :**

- 计算  $R' = \sigma \cdot G - \alpha \cdot Q_A$
- 验证等式:  $\text{hash}(m || R') \equiv \alpha$ ;
- 如果等式成立输出1, 否则输出0。



## \*Project22: research report on MPT

### #一、概述

Merkle Patricia Tree (又称为 Merkle Patricia Trie) 是一种经过改良的、融合了默克尔树和前缀树两种树结构优点的数据结构，是以太坊中用来组织管理账户数据、生成交易集合哈希的重要数据结构。

MPT 树有以下几个作用：

- 存储任意长度的 key-value 键值对数据；
- 提供了一种快速计算所维护数据集哈希标识的机制；
- 提供了快速状态回滚的机制；
- 提供了一种称为默克尔证明的证明方法，进行轻节点的扩展，实现简单支付验证；

### 1 前缀树

前缀树（又称字典树），用于保存关联数组，其键（key）的内容通常为字符串。前缀树节点在树中的位置是由其键的内容所决定的，即前缀树的 key 值被编码在根节点到该节点的路径中。

优势：

相比于哈希表，使用前缀树来进行查询拥有共同前缀 key 的数据时十分高效，例如在字典中查找前缀为 pre 的单词，对于哈希表来说，需要遍历整个表，时间效率为  $O(n)$ ；然而对于前缀树来说，只需要在树中找到前缀为 pre 的节点，且遍历以这个节点为根节点的子树即可。

但是对于最差的情况（前缀为空串），时间效率为  $O(n)$ ，仍然需要遍历整棵树，此时效率与哈希表相同。

相比于哈希表，在前缀树不会存在哈希冲突的问题。

劣势：

直接查找效率低下：

前缀树的查找效率是  $O(m)$ ， $m$  为所查找节点的 key 长度，而哈希表的查找效率为  $O(1)$ 。且一次查找会有  $m$  次 I/O 开销，相比于直接查找，无论是速率、还是对磁盘的压力都比较大。

可能会造成空间浪费：

当存在一个节点，其 key 值内容很长（如一串很长的字符串），当树中没有与他相同前缀的分支时，为了存储该节点，需要创建许多非叶子节点来构建根节点到该节点间的路径，造成了存储空间的浪费。

### 2 默克尔树

Merkle 树是由计算机科学家 Ralph Merkle 在很多年前提出的，并以他本人的名字来命名，由于在比特币网络中用到了这种数据结构来进行数据正确性的验证，在这里简要地介绍一下 merkle 树的特点及原理。

在比特币网络中，merkle 树被用来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹。此外，由于 merkle 树的存在，使得在比特币这种公链的场景下，扩展一种“轻节点”实现简单支付验证变成可能，关于轻节点的内容，将会下文详述。

特点：

默克尔树是一种树，大多数是二叉树，也可以多叉树，无论是几叉树，它都具有树结构的所有特点；

默克尔树叶子节点的 value 是数据项的内容，或者是数据项的哈希值；

非叶子节点的 value 根据其孩子节点的信息，然后按照 Hash 算法计算而得出的；

原理：

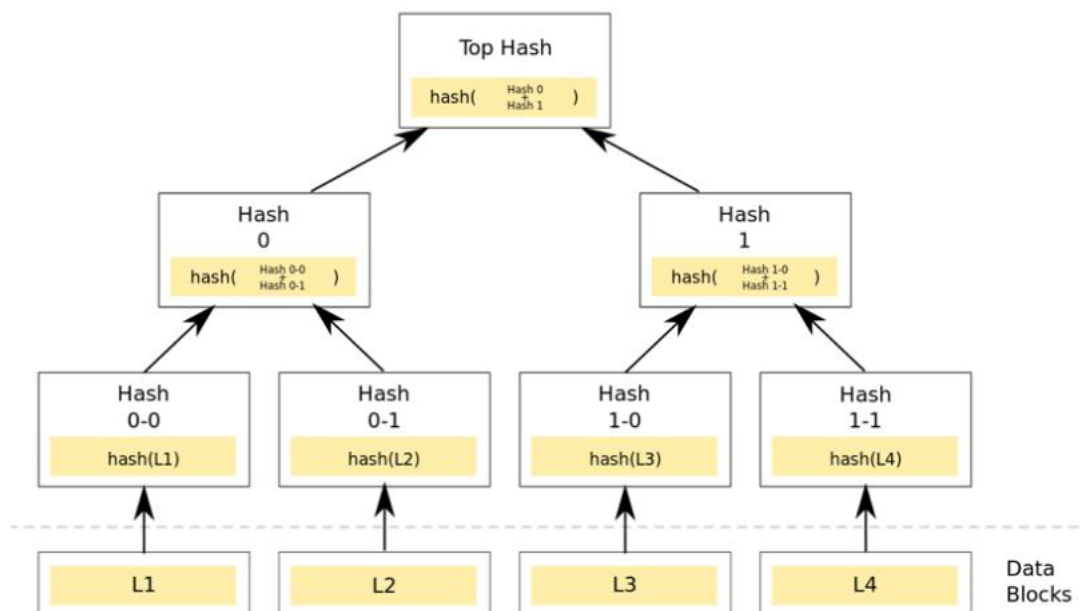
在比特币网络中，merkle 树是自底向上构建的。在下图的例子中，首先将 L1-L4 四个单元数据哈希化，然后将哈希值存储至相应的叶子节点。这些节点是 Hash0-0, Hash0-1, Hash1-0, Hash1-1

将相邻两个节点的哈希值合并成一个字符串，然后计算这个字符串的哈希，得到的就是这两个节点的父节点的哈希值。

如果该层的树节点个数是单数，那么对于最后剩下的树节点，这种情况就直接对它进行哈希运算，其父节

点的哈希就是其哈希值的哈希值（对于单数个叶子节点，有着不同的处理方法，也可以采用复制最后一个叶子节点凑齐偶数个叶子节点的方式）。循环重复上述计算过程，最后计算得到最后一个节点的哈希值，将该节点的哈希值作为整棵树的哈希。

若两棵树的根哈希一致，则这两棵树的结构、节点的内容必然相同。



优势

1 快速重哈希：

默克尔树的特点之一就是当树节点内容发生变化时，能够在前一次哈希计算的基础上，仅仅将被修改的树节点进行哈希重计算，便能得到一个新的根哈希用来代表整棵树的状态。

2 轻节点扩展：

采用默克尔树，可以在公链环境下扩展一种“轻节点”。轻节点的特点是对于每个区块，仅仅需要存储约 80 个字节大小的区块头数据，而不存储交易列表，回执列表等数据。然而通过轻节点，可以实现在非信任的公链环境中验证某一笔交易是否被收录在区块链账本的功能。这使得像比特币，以太坊这样的区块链能够运行在个人 PC，智能手机等拥有小存储容量的终端上。

劣势：

1 存储空间开销大：

## #二、结构设计

1 节点分类

前缀树无论是查询操作，还是对数据的增删改，不仅效率低下，且存储空间浪费严重。故，在以太坊中，为 MPT 树新增了几种不同类型的树节点，以尽量压缩整体的树高、降低操作的复杂度。

MPT 树中，树节点可以分为以下四类：空节点、分支节点、叶子节点、扩展节点

空节点：用来表示空串

分支节点：用来表示 MPT 树中所有拥有超过 1 个孩子节点以上的非叶子节点，其定义如下所示：

```

1 type fullNode struct {
2     Children [17]node // Actual trie node data to encode/decode (needs custom e
3     flags    nodeFlag
4 }
5 // nodeFlag contains caching-related metadata about a node.
6 type nodeFlag struct {
7     hash hashNode // cached hash of the node (may be nil)
8     gen  uint16    // cache generation counter
9     dirty bool      // whether the node has changes that must be written to the data
10 }

```

叶子节点&&扩展节点:

```

1 type shortNode struct {
2     Key    []byte
3     Val    node
4     flags nodeFlag
5 }

```

其中关键的字段为:

Key: 用来存储属于该节点范围的 key;

Val: 用来存储该节点的内容;

其中 Key 是 MPT 树实现树高压压缩的关键!

## 2 安全的 MPT

以上介绍的 MPT 树, 可以用来存储内容为任何长度的 key-value 数据项。倘若数据项的 key 长度没有限制时, 当树中维护的数据量较大时, 仍然会造成整棵树的深度变得越来越深, 会造成以下影响:

查询一个节点可能会需要许多次 IO 读取, 效率低下;

系统易遭受 Dos 攻击, 攻击者可以通过在合约中存储特定的数据, “构造” 一棵拥有一条很长路径的树, 然后不断地调用 SLOAD 指令读取该树节点的内容, 造成系统执行效率极度下降;

所有的 key 其实是一种明文的形式进行存储;

为了解决以上问题, 在以太坊中对 MPT 再进行了一次封装, 对数据项的 key 进行了一次哈希计算, 因此最终作为参数传入到 MPT 接口的数据项其实是 (sha3(key), value)

优势:

1 传入 MPT 接口的 key 是固定长度的 (32 字节), 可以避免出现树中出现长度很长的路径;

劣势:

1 每次树操作需要增加一次哈希计算;

2 需要在数据库中存储额外的 sha3(key) 与 key 之间的对应关系;

## #三、基本操作

### 1 Get

一次 Get 操作的过程为:

将需要查找 Key 的 Raw 编码转换成 Hex 编码, 得到的内容称之为搜索路径;

从根节点开始搜寻与搜索路径;

内容一致的路径;

若当前节点为叶子节点, 存储的内容是数据项的内容, 且搜索路径的内容与叶子节点的 key 一致, 则表示找到该节点; 反之则表示该节点在树中不存在。

若当前节点为扩展节点, 且存储的内容是哈希索引, 则利用哈希索引从数据库中加载该节点, 再将搜索路

径作为参数，对新解析出来的节点递归地调用查找函数。

若当前节点为扩展节点，存储的内容是另外一个节点的引用，且当前节点的 key 是搜索路径的前缀，则将搜索路径减去当前节点的 key，将剩余的搜索路径作为参数，对其子节点递归地调用查找函数；若当前节点的 key 不是搜索路径的前缀，表示该节点在树中不存在。

若当前节点为分支节点，若搜索路径为空，则返回分支节点的存储内容；反之利用搜索路径的第一个字节选择分支节点的孩子节点，将剩余的搜索路径作为参数递归地调用查找函数。

## 2 Insert

插入操作也是基于查找过程完成的，一个插入过程为：

根据 3.1 中描述的查找步骤，首先找到与新插入节点拥有最长相同路径前缀的节点，记为 Node；

若该 Node 为分支节点：

- (1) 剩余的搜索路径不为空，则将新节点作为一个叶子节点插入到对应的孩子列表中；
- (2) 剩余的搜索路径为空(完全匹配)，则将新节点的内容存储在分支节点的第 17 个孩子节点项中(Value)；

若该节点为叶子 / 扩展节点：

- (1) 剩余的搜索路径与当前节点的 key 一致，则把当前节点 Val 更新即可；
- (2) 剩余的搜索路径与当前节点的 key 不完全一致，则将叶子 / 扩展节点的孩子节点替换成分支节点，将新节点与当前节点 key 的共同前缀作为当前节点的 key，将新节点与当前节点的孩子节点作为两个孩子插入到分支节点的孩子列表中，同时当前节点转换成了一个扩展节点（若新节点与当前节点没有共同前缀，则直接用生成的分支节点替换当前节点）；

若插入成功，则将被修改节点的 dirty 标志置为 true，hash 标志置空（之前的结果已经不可能用），且将节点的诞生标记更新为现在；

## 3 Delete

删除操作与插入操作类似，都需要借助查找过程完成，一次删除过程为：

根据 3.1 中描述的查找步骤，找到与需要插入的节点拥有最长相同路径前缀的节点，记为 Node；

若 Node 为叶子 / 扩展节点：

- (1) 若剩余的搜索路径与 node 的 Key 完全一致，则将整个 node 删除；
- (2) 若剩余的搜索路径与 node 的 key 不匹配，则表示需要删除的节点不存于树中，删除失败；
- (3) 若 node 的 key 是剩余搜索路径的前缀，则对该节点的 Val 做递归的删除调用；

若 Node 为分支节点：

- (1) 删除孩子列表中相应下标标志的节点；
- (2) 删除结束，若 Node 的孩子个数只剩下一个，那么将分支节点替换成一个叶子 / 扩展节点；

若删除成功，则将被修改节点的 dirty 标志置为 true，hash 标志置空（之前的结果已经不可能用），且将节点的诞生标记更新为现在；

## 4 Update

更新操作就是 3.2Insert 与 3.3Delete 的结合。当用户调用 Update 函数时，若 value 不为空，则隐式地转为调用 Insert；若 value 为空，则隐式地转为调用 Delete，故在此不再赘述。

## 5 Commit

Commit 函数提供将内存中的 MPT 数据持久化到数据库的功能。在第一章中我们提到的 MPT 具有快速计算所维护数据集哈希标识以快速状态回滚的能力，也都是在该函数中实现的。

在 commit 完成后，所有变脏的树节点会重新进行哈希计算，并且将新内容写入数据库；最终新的根节点哈希将被作为 MPT 的最新状态被返回。

一次 MPT 树提交是一个递归调用的过程，在介绍 MPT 提交过程之前，我们首先介绍单个节点是如何进行哈希计算和存储的。